




Introduction to Recursion

WEDNESDAY, APRIL 16

An introduction to the interesting world of recursion, where a function can call itself!

-  Readings: [Text](#) 7.1-7.3
-  [Lecture quiz on Canvas](#)
-  [Lecture video on Canvas](#)

Lecture Video

Click to sign in and play video

Contents

1. Don't panic!
2. Overview
3. Our First Recursive Function (and Stack Overflow)
4. Recursion vs. Iteration
5. Factorial
6. Common Pitfall #1
7. Common Pitfall #2
8. Random Interlude: Unsigned Integers (*Supplementary*)
9. Palindromes
10. Common Pitfall #3
11. String Printing
12. Reverse String Printing
13. Wrapper Functions
14. Common Pitfall #4
15. **Sneak Preview!** Coin Flips
16. What's next?

17. Exam Prep

Don't panic!

Let me say that a bit more loudly for anyone who felt like today's lecture was a totally wild, mind-bending ride:

DON'T PANIC!

Recursion is *crazy* the first time you see it in code! Don't feel discouraged if today's lecture felt super weird or foreign or mystical or difficult. This is a tough topic. It will take lots of practice, but eventually, writing recursive functions will feel like second nature.

Overview

Today, we made our foray into recursion, examining some basic examples of functions that call themselves. We saw that there are two critical components to recursive functions:

1. **Base Case.** Include some sort of terminating condition that returns the result for some canonical case where we know the answer immediately.
2. **Recursive Call.** Decompose the current input into subproblems, one of which involves making a recursive call to the function you're writing. Make sure your input is approaching the base case (not growing infinitely in the opposite direction)!

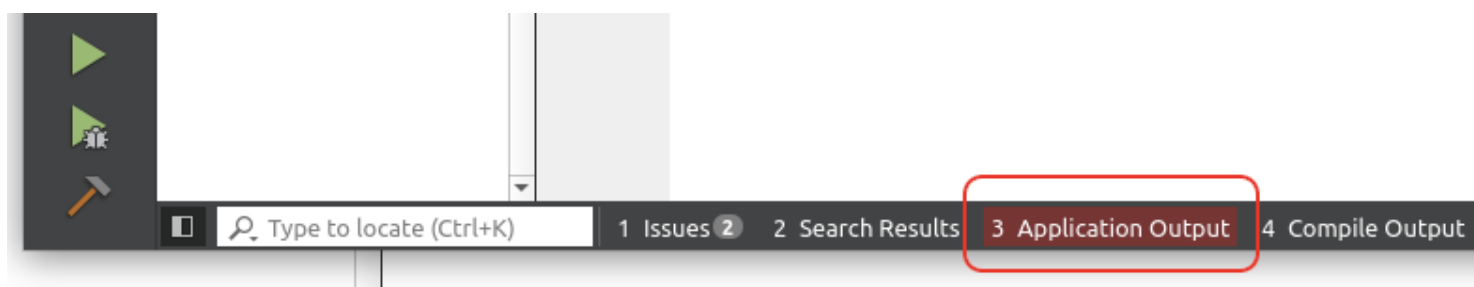
Our First Recursive Function (and Stack Overflow)

We started with this simple recursive function that doesn't accomplish anything interesting:

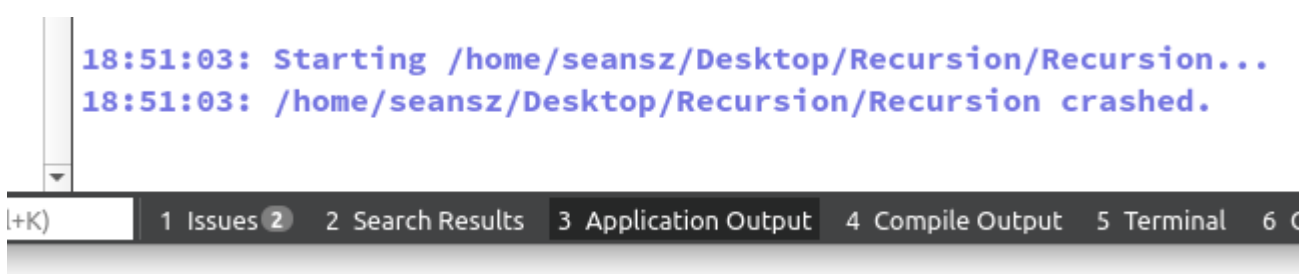
```
void foo()
{
    foo();
}

int main()
{
    foo();
    return 0;
}
```

We dissected its behavior and saw that the infinite recursion fills up our program stack with recursive calls, causing a stack overflow. In the Qt Creator, the terminal window just flashed briefly, and the fact that our program crashed might not have been immediately apparent, but the "Application Output" tab at the bottom of the window was flashing red:



Clicking that tab gave us a rather bland and not-nearly-alarming-enough message indicating that the program had crashed:



We then augmented the program to give us some indication of how many recursive calls we were able to make before crashing:

```
#include <iostream>
#include "console.h"
using namespace std;

void foo(int n)
{
    // We used this mod trick because printing to the screen is such a slow
    // operation. When we tried printing n in every single function call, it
    // was taking way too long. (On the computer in my office, it takes about
    // nine minutes for the program to crash when printing every single n.
    // When printing n once per 1000 calls, it takes less than one second to
    // crash.

    if (n % 1000 == 0)
    {
        cout << n << endl;
    }

    foo(n + 1);
}

int main()
{
    foo(0);
    return 0;
}
```

output:

```
...
261945
261946
261947
261948
261949
19:06:18: /home/seansz/Desktop/Recursion/Recursion crashed.
```

Finally, we added a base case to the function and re-wrote it in such a way that each recursive call was working toward that base case:

```
#include <iostream>
#include "console.h"
using namespace std;

void foo(int n)
{
    if (n == 0)
    {
        cout << "Blastoff!" << endl;
        return;
    }

    cout << n << "..." << endl;
    foo(n - 1);
}

int main()
{
    foo(5);
    return 0;
}
```

output:

```
5...
4...
3...
2...
1...
Blastoff!
```

Recursion vs. Iteration

Note that most of the recursive functions we saw in class today could have been implemented iteratively (i.e., using for-loops that iterate through the problems, rather than using recursion), but we played with the recursive implementations anyway because they offer a fairly gentle introduction to the idea of recursion (insofar as *any*

introduction to recursion could be considered gentle...). We'll see more useful examples of recursion throughout the quarter.

Factorials

We then explored how to write a recursive factorial function. We started by observing that for $n > 0$, $n!$ has at least one other factorial embedded in it: $(n - 1)!$. We capitalized on that to write a mathematical version of $n!$ recursively, then translated that into code:

Recursion - Factorial - Decomposition

SS By Sean Szumlanski
April 16, 2025

Present

From there we saw how individual **stack frames** piled up as we made recursive calls to our function, starting with `factorial(5)` :

Recursion - Factorial - Call Stack

SS By Sean Szumlanski
April 16, 2025

Present

I also presented a visualization of the recursive calls that showed us diving deeper into recursive calls and then returning from them:

Recursion - Factorial - Nesting (Spring 2025)

SS By Sean Szumlanski
April 16, 2025

Present

Our final program is as follows:

```
#include <iostream>
#include "console.h"
using namespace std;

int factorial(int n)
{
    if (n == 0)
    {
        return 1;
    }

    return n * factorial(n - 1);
}

int main()
{
    cout << factorial(5) << endl;
    return 0;
}
```

output:

120

Common Pitfall #1

Over the years, I have seen a **lot** of people leave return statements off their functions, like so:

```
int factorial(int n)
{
    if (n == 0)
    {
        return 1;
    }

    n * factorial(n - 1); // NOoooooOooooooo! :(
}
```

This won't be a problem when working with the Qt Creator this quarter because our project flags stop you from compiling if an `int` function doesn't return a value. By default, however, a C++ compiler *will* let you compile such a function, and so it's important to be aware of that moving forward.

Common Pitfall #2

With the factorial example, we also discussed the importance of our code actually accounting for all possible valid inputs. Although factorials are not defined for negative integers, $0!$ *is* defined. So, the following code would be insufficient. If we called `factorial(0)`, we would get stack overflow instead of a valid result:

```
int factorial(int n)
{
    if (n == 1) // NOoooooOooooooo! :( What about n == 0?
    {
        return 1;
    }

    return n * factorial(n - 1);
}
```

Random Interlude: Unsigned Integers (*Supplementary*)

This is supplementary information. You don't need to know this right now.

(*Not mentioned in class.*) Above, I'm inclined not to worry about what happens if someone calls `factorial()` with a negative integer, since the function needn't be defined for negative integers. However, if you wanted to restrict inputs to non-negative integers, there's a data type in C++ for that called `unsigned int`:

```
unsigned int factorial(unsigned int n)
{
    if (n == 0)
    {
        return 1;
    }

    return n * factorial(n - 1);
}
```

Palindromes

The next recursive function we built detected whether or not a string was a palindrome. Recall that a **palindrome** is a string that is the same forward as it is backward.

We started by examining the structure of the problem and observing that by peeling away the first and last character of a string (if those characters matched), we uncovered a subproblem that could be passed to our function recursively: namely, whether the remaining portion of the string was, itself, a palindrome.

For an exploration of the recursive framing of this problem, see the following Prezi:

Recursion - Palindrome - Decomposition

ss By Sean Szumlanski
April 16, 2025

Present

For a visualization of the stack frames associated with all the recursive calls that flow from a call to `isPalindrome("racecar")`, see the following:

Recursion - Palindrome - Call Stack

ss By Sean Szumlanski
April 16, 2025

Present

Our final version of the palindrome code is as follows:

```

#include <iostream>
#include "console.h"
#include "SimpleTest.h"
using namespace std;

bool isPalindrome(string s)
{
    // All strings of length 1 are palindromes, as is the empty string.
    if (s.length() <= 1)
    {
        return true;
    }

    if (s[0] != s[s.length() - 1])
    {
        return false;
    }

    // Peel off the first and last character and pass the remaining string
    // to isPalindrome() recursively. We needn't use a new string variable
    // here. I did that in class only to clarify what was going on. The
    // more conventional approach is to pass s.substr(...) to isPalindrome()
    // directly.
    string sub = s.substr(1, s.length() - 2);
    return isPalindrome(sub);
}

PROVIDED_TEST("various palindromes")
{
    EXPECT_EQUAL(isPalindrome("racecar"), true);
    EXPECT_EQUAL(isPalindrome("kayak"), true);
    EXPECT_EQUAL(isPalindrome("civic"), true);
    EXPECT_EQUAL(isPalindrome("radar"), true);
    EXPECT_EQUAL(isPalindrome("rotor"), true);
    EXPECT_EQUAL(isPalindrome("step on no pets"), true);
    EXPECT_EQUAL(isPalindrome("b"), true);
    EXPECT_EQUAL(isPalindrome(""), true);
}

int main()
{
    runSimpleTests(ALL_TESTS);
    return 0;
}

```

Common Pitfall #3

Note that the test cases above would not give any indication that the following version of `isPalindrome()` is broken:

```

bool isPalindrome(string s)
{
    if (s.length() <= 1)
    {
        return true;
    }

    // ERROR: Not checking whether first and last characters match.
    // This function returns true for all possible inputs!

    string sub = s.substr(1, s.length() - 2);
    return isPalindrome(sub);
}

```

The fact that this version would pass all the test cases above highlights the importance of testing not just whether a function like this returns `true` when it's expected to, but also that it returns `false` as intended.

String Printing

We then took a moment to work on an in-class exercise. The challenge I presented was to write a recursive function that prints a string, with the restriction that any given call can only print a single character. We came up with the following:

```
#include <iostream>
#include "console.h"
using namespace std;

void printString(string s)
{
    if (s.length() == 0)
    {
        cout << endl;
        return;
    }

    cout << s[0];

    // The string passed recursively has s[0] shaved off.
    printString(s.substr(1, s.length() - 1));
}

int main()
{
    // Calling twice to ensure proper placement of line break in output.
    printString("hello");
    printString("hello");

    return 0;
}
```

output:

```
hello
hello
```

(*Important note!*) If we pass only one parameter to `substr()`, we get a substring starting at that index and going to the end of the string. So, in the code above, we could use `s.substr(1)` in place of `s.substr(1, str.length() - 1)`.

Reverse String Printing

I then pointed out that we could modify the code above to print our string in reverse. To do that, we simply swap the order of our `cout` and `printStringReverse()` lines in the original recursive function, and let the program stack take care of reversing the string for us:

```
#include <iostream>
#include "console.h"
using namespace std;

void printStringReverse(string s)
{
    if (s.length() == 0)
    {
        cout << endl; // This is problematic.
        return;
    }

    // Swapped order of lines below.
    printStringReverse(s.substr(1));
    cout << s[0];
}

int main()
{
    // Calling twice to ensure proper placement of line break in output.
    printStringReverse("hello");
    printStringReverse("hello");

    return 0;
}
```

output:

```
olleh
olleh
```

Notice that the output actually places the newline character *before* each reversed string, not after. We resolve that issue in the following section.

(*Not mentioned in class.*) An alternative approach to this problem would be to print the last character of the string in

each recursive call, like so:

```
void printStringReverse(string s)
{
    if (s.length() == 0)
    {
        cout << endl;
        return;
    }

    cout << s[s.length() - 1];
    printStringReverse(s.substr(0, s.length() - 1));
}
```

Wrapper Functions

The base case of our recursive `printStringReverse()` function is not the right place to print `endl`. We hit the base case before *any* other characters have printed to the screen, so if we place the `endl` there, we print our newline character *before* we print out the string in reverse.

We also don't want to tell someone who calls our function that they have to print a newline character themselves. It's a natural part of the desired behavior of a string printing function like this one, and so we should handle that ourselves.

(*Key take-away!*) One way to accomplish this is through the use of a **wrapper function** -- a function that acts as a sort of gateway to our core function, but takes care of any setup or tear-down that needs to happen before or after that core function is executed. Here's what we did in class:

```
#include <iostream>
#include "console.h"
using namespace std;

void printStringReverseForRealsies(string s)
{
    if (s.length() == 0)
    {
        return;
    }

    printStringReverseForRealsies(s.substr(1));
    cout << s[0];
}

void printStringReverse(string s)
{
    printStringReverseForRealsies(s);
    cout << endl;
}

int main()
{
    // Calling twice to ensure proper placement of line break in output.
    printStringReverse("hello");
    printStringReverse("hello");

    return 0;
}
```

output:

```
olleh
olleh
```

(*Not mentioned in class.*) The name `printStringReverseForRealsies()` is highly unconventional. It would be more common to refer to that so-called helper function as `printStringReverseHelper()` or something similar.

(*Not mentioned in class.*) In the event that the *original* string passed to our function is empty, we might not want to print a newline character at all. To accomplish that, we could add a conditional statement to the `printStringReverse()` wrapper function.

Common Pitfall #4

When creating the wrapper function above and renaming the original `printString()` function to `printStringForRealsies()`, I pointed out one of the cardinal sins of modifying a recursive function name:

forgetting to update the recursive call(s) within that function to the new name. The place where this probably comes up the most is in copying and pasting recursive functions you've already written and then making minor tweaks to them to achieve new functionality.

Sneak Preview! Coin Flips

At the end of class, I talked briefly about the problem of printing out all possible outcomes from flipping a coin n times. For example, if we flip a coin twice, our outcomes are as follows (where 'H' is for flipping heads and 'T' is for flipping tails):

HH, HT, TH, TT

This is a bit obnoxious to achieve iteratively, but can be achieved elegantly with just a few lines of code if we use recursion. We will explore this in more depth in our next lecture, but in the meantime, if you want to look ahead at how that works, here's the Prezi, which includes a recursive solution to this problem:

Recursion - Coin Flip

SS By Sean Szumlanski
April 16, 2025

Present

What's next?

On Friday, we will continue our exploration of recursion and start exploring examples of some very interesting problems we can solve with this technique!

Exam Prep

1. Enrich your knowledge of the material we covered today by reviewing all the sections of today's notes labeled "(*Not mentioned in class.*)" Be sure to review all the "Common Pitfall" sections, as well.

2. After reviewing today's examples, take at least a 30-minute break, then try to code up each of the following recursive functions from scratch, without referring to the notes:

(a) `int factorial(int n)`

(b) `bool isPalindrome(string s)`

(c) `void printString(string s)`

3. (*High priority! Definitely do this one!*) The `printString()` function from today's lecture is somewhat inefficient because it is repeatedly constructing new strings with the `s.substr()` call. Write a different version of the function that takes a *reference* to a string and passes that reference to all recursive calls to save time and space. Add a second parameter to the function (an integer parameter) to keep track of how far along we are in the string, like so:

```

#include <iostream>
#include "console.h"
using namespace std;

void printStringHelper(string& s, int k)
{
    if ( ??? )
    {
        cout << endl;
        return;
    }

    cout << s[ ??? ];
    printStringHelper(s, ??? );
}

void printString(string& s)
{
    printStringHelper(s, ??? );
}

int main()
{
    string s = "hello";
    printString(s);
    printString(s);

    return 0;
}

```

Highlight for solution:

```

#include <iostream>
#include "console.h"
using namespace std;

void printStringHelper(string& s, int k)
{
    if (k == s.length())
    {
        cout << endl;
        return;
    }

    cout << s[k];
    printStringHelper(s, k + 1);
}

void printString(string& s)
{
    printStringHelper(s, 0);
}

int main()
{
    string s = "hello";
    printString(s);
    printString(s);

    return 0;
}

```

4. Modify `printStringReverse()` and `printStringReverseHelper()` so that the program below prints a newline *after* our string has been printed in reverse, but with the following restrictions:

- (a) You cannot add new lines to `printStringReverse()` or remove lines from that function. You can, however, modify the one line in that function.
- (b) You cannot rearrange the order of the last two lines of the `printStringReverseHelper()` function. You *must* call `printStringReverseHelper()` recursively before executing the `cout` statement. See the comment in the code below for more detail.

```
#include <iostream>
#include "console.h"
using namespace std;

void printStringReverseHelper(string s)
{
    if (s.length() == 0)
    {
        return;
    }

    // You can modify the following lines, but do not swap their order. So, the
    // cout statement must come after the printStringReverseHelper() statement.
    // You can, however, add lines before or after the following, or even change
    // how the printStringReverseHelper() function is called.
    printStringReverseHelper(s.substr(1));
    cout << s[0];
}

void printStringReverse(string s)
{
    // You can modify the following line, but it must call printStringReverseHelper(),
    // and you cannot add any other lines of code to this function.
    printStringReverseHelper(s);
}

int main()
{
    string s = "hello";
    printStringReverse(s);
    printStringReverse(s);

    return 0;
}
```

output:

olleholleh

Highlight for hint:

There's a way to do this that involves passing an extra parameter to the helper function.

Highlight for solution:

```

#include <iostream>
#include "console.h"
using namespace std;

void printStringReverseHelper(string s, bool isOriginalCall)
{
    if (s.length() == 0)
    {
        return;
    }

    printStringReverseHelper(s.substr(1), false);
    cout << s[0];

    // This should only be true for the very first call to this function, which
    // is exactly where we want to print the newline character (after returning
    // from the other function calls).

    if (isOriginalCall)
    {
        cout << endl;
    }
}

void printStringReverse(string s)
{
    // Pass true to this function to convey that it is the originating call for
    // this string.

    printStringReverseHelper(s, true);
}

int main()
{
    string s = "hello";
    printStringReverse(s);
    printStringReverse(s);

    return 0;
}

```

output:

```

olleh
olleh

```

5. (High priority! Definitely do this one!) Our `isPalindrome()` function from class peels off the first and last character of our string before making a recursive call. There is, however, an alternative approach: we could start at the *middle* of our string and extract characters from there before making our recursive calls. For example, a call to `isPalindrome("racecar")` would make a recursive call to `isPalindrome("raccar")` (with the `e` removed from the middle of the string. Code up this version of the function, which has some fun gotchas and complexities to discover and deal with along the way.

6. What critical shortcoming does the following suite of test cases for our recursive `isPalindrome()` function suffer from?

```

PROVIDED_TEST("various palindromes")
{
    EXPECT_EQUAL(isPalindrome("racecar"), true);
    EXPECT_EQUAL(isPalindrome("kayak"), true);
    EXPECT_EQUAL(isPalindrome("civic"), true);
    EXPECT_EQUAL(isPalindrome("radar"), true);
    EXPECT_EQUAL(isPalindrome("rotor"), true);
    EXPECT_EQUAL(isPalindrome("step on no pets"), true);
    EXPECT_EQUAL(isPalindrome("b"), true);
    EXPECT_EQUAL(isPalindrome(""), true);
}

```

Highlight for answer:

There are no test cases where we expect the function to return **false**! If our function simply returned **true** in all cases, it would pass this test case suite with flying colors despite failing to even attempt to solve the problem in any meaningful way.

7. Be sure to trace through the `coinFlip()` function given in the final Prezi above to see if you can make sense of how it's working. You might want to use your debugger to see how `s` is changing with each successive function call, or introduce various `cout` statements that print information about all the stack frames as the function makes its recursive calls.

8. Be sure to read the textbook for further explanation, examples, and practice problems related to recursion.

All course materials © Stanford University 2024. This content is protected and may not be shared, uploaded, or distributed.

Website programming by Julie Zelenski with modifications by Sean Szumlanski • Styles adapted from Chris Piech • This page last updated 2025-Apr-16